SANDIA REPORT

SAND2015-6031
Unlimited Release
Printed July 2015

Motivation and Design of the Sirocco Storage System, Version 1.0

Matthew L. Curry, H. Lee Ward, Geoff Danielson

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2015-6031
Unlimited Release
Printed July 2015

Motivation and Design of the Sirocco Storage System, Version 1.0

Matthew L. Curry
Center for Computing Research
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-1319
mlcurry@sandia.gov

H. Lee Ward
Center for Computing Research
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-1319
lee@sandia.gov

Geoff Danielson
Hewlett-Packard Company
Geoffrey.Danielson@hp.com

Abstract

Sirocco is a massively parallel, high performance storage system for the exascale era. It emphasizes client-to-client coordination, low server-side coupling, and free data movement to improve resilience and performance. Its architecture is inspired by peer-to-peer and victim-cache architectures. By leveraging these ideas, Sirocco natively supports several media types, including RAM, flash, disk, and archival storage, with automatic migration between levels.

Sirocco also includes storage interfaces and support that are more advanced than typical block storage. Sirocco enables clients to efficiently use key-value storage or block-based storage with the same interface. It also provides several levels of transactional data updates within a single storage command, including full ACID-compliant updates. This transaction support extends to updating several objects within a single transaction. Further support is provided for concurrency control, enabling greater performance for workloads while providing safe concurrent modification.

By pioneering these and other technologies and techniques in the storage system, Sirocco is poised to fulfill a need for a massively scalable, write-optimized storage system for exascale systems.

This is version 1.0 of a document reflecting the current and planned state of Sirocco. Further versions of this document will be accessible at `http://www.cs.sandia.gov/Scalable_IO/sirocco`.

Contents

Introduction 7
Sirocco Design Principles 8
System Overview 9
The Storage Swarm 9
Read/Location Protocols 13
Conclusion 15
References 16

This page intentionally left blank.

Introduction

Evidence is mounting that current solutions to file storage are going to be inadequate for exascale systems. The hardware composition of storage systems are going through fundamental changes driven by changing economics of storage devices. Pure disk-based systems are no longer capable of sustaining the I/O bandwidth required to write full-system checkpoints, forcing designers to include heterogeneous, task-specific storage technologies into system designs. Further, exploding core counts are increasing the numbers of clients requiring access to storage.

We believe that storage system software needs a radical redesign to maintain high write bandwidth for checkpoint-restart, a critical capability for large scale simulation. Sirocco is a new design for a write-optimized parallel storage service that uses novel techniques to address the current challenges of storage for massively parallel workloads.

Sirocco is a parallel storage system for extreme scale computing. It is inspired by the loosely coupled semantics of unstructured peer-to-peer file sharing systems. By leveraging a peer-based caching architecture throughout the storage system, clients are free to target appropriate storage servers based on such metrics as network performance, load, capacity, and media characteristics, without regard to assigned data location. This design enables local (i.e., within file system client, storage server, or job) decision making instead of global state maintenance, allowing unrelated tasks to proceed without unnecessary contention. We believe that the result will be superior, uncompromising write performance.

Sirocco Design Principles

There are a small number of guiding principles for Sirocco design.

1. There is no central index that determines where a piece of data must be stored. Clients of the storage system are allowed to place data within any store of the system they can reach. As a consequence, the location of a required piece of data may not be known at the time it is needed.

2. Data will be continually moving within the system to ensure longevity, integrity, and system health. Replicas will be created and destroyed, and servers will eject data into more durable stores. Clients will not be notified of these events. A related benefit of this is that growing and shrinking a Sirocco store is trivial.

3. Sirocco's design emphasizes scalability over legacy support. Support for legacy storage system semantics, like POSIX, are required; however, overall system scalability should not be harmed by POSIX considerations. Indeed, more scalable storage semantics should see better performance over Sirocco than POSIX.

4. Heterogeneous media (including temporary RAM-based stores, flash-based burst buffers, disk, tape, and others as they become available) should be supported transparently, with symmetric APIs for data access.

5. All data are not created equal, and some forms need higher resilience guarantee than others. Clients should be able to determine the level at which Sirocco protects data.

6. All server-side operations should be as scalable as possible, particularly when running on real-world, faulty hardware. Therefore, servers should not couple during any operation.

System Overview

Sirocco is designed to be the storage component for the Lightweight File System, an architecture that provides basic storage services necessary to all applications, while leaving other functionality to be provided by libraries [7]. Examples of "optional" services provided by libraries include naming, consistency, metadata, and interfaces. Below are a list of features provided natively by Sirocco.

The Storage Swarm

One of the central defining features of Sirocco is the non-static, Quality-of-Service-oriented organization of the storage servers. This is referred to as an unstructured peer-to-peer organization. As opposed to structures like distributed hash tables [8], where a structured overlay network is employed to enable routing, data distribution, and location, Sirocco eschews structure to instead focus on maximizing overall network performance among stores. This is accomplished by choosing the closest, highest-performing peers available as neighbors, and using those as storage targets and next hops for location requests.

Sirocco uses a two-stage method for building a useful overlay network. For the first stage, we are currently evaluating the Self-Aware P2P (SAP2P) overlay, an adaptive and locality-aware peer-to-peer membership protocol [6]. It includes distance-optimized links, using the lowest known latency links to form local neighbor connections to nearby nodes, as well as a small number of random links to ensure connectivity. The second stage is driven by locality as well as node needs– Addresses for nodes that have necessary resources (e.g., durable stores or large amounts of free space) are preferred.

This comprehensive overlay is used for a variety of purposes, including data migration and replication. One of the design criteria that SAP2P claims to satisfy is churn resilience: That is, significant numbers of members should be able to join and leave the network, and the overlay should be able to avoid partitioning.

One reason that churn resilience is so important is because Sirocco's design lends itself to an interesting use case. To accelerate checkpoint workloads, one can use an allocation of compute nodes as write back caches for checkpoint workloads. This is a similar technique to the use of a burst buffer. We showed the efficacy of using an application-specific client to achieve massive checkpoint speedup using Sirocco compute section allocations [3]. In order to facilitate this use, Sirocco should be able to handle near-simultaneous joining and departure of hundreds or thousands of servers in short time frames.

Name Space

Sirocco is a parallel object-based storage system that provides primitives to enable higher-level parallel file systems. As such, it does not provide human friendly names and constructs, but instead has an ID-based name space. This object name space is an implementation of the Advanced Storage Group (ASG) interface [4], which was partially developed for Sirocco in collaboration with Argonne National Laboratory. This section describes how clients use the name space, and storage within it.

The name space is made up of three 64-bit values. These values denote a container ID, object ID, and fork ID, which can be expressed as $\langle \text{container}, \text{object}, \text{fork} \rangle$. Loosely, container IDs usually map to file systems within the storage system, object IDs map to files, and fork IDs map to data forks within a file. The hierarchy and relationships are static; forks cannot move between objects, for example.

Each fork contains a key-value (KV) store, where the key is 64 bits, and the value (or record) is a variable-length record that can theoretically be up to 2^{64} bytes long. Currently, a value is practically limited to 2^{32} bytes. Each value is atomically modified in its entirety; one cannot partially update a record. As will be discussed, the ASG interface supports KV operations and traditional bulk operations with the same functions.

Sirocco reserves forks within the name space for security information. Each container x such that $x \neq 0$ has security information recorded in the KV store located in $\langle 0,0,0 \rangle$, record x. Each object y in container x has security information stored in the KV store $\langle x,0,0 \rangle$, record y. Each fork z in object y in container x has security information stored in record z of $\langle x,y,0 \rangle$. Access to the records within a fork is protected by the security attributes of the fork. If not present, attributes are inherited from the object or container.

Each record has a user-modifiable attributed called the update ID. The update ID is a logical clock that is expected to increase after each write. This mechanism is used to determine the most recent version of a particular record, given that multiple versions of the record may exist on different stores in the storage system. In particular, Sirocco will use the update ID to reconcile two instances of a record during migration from one server to another, or while searching for data: If the update IDs are different for a record, the largest update ID determines which record is correct. The client is expected to know (or determine) an appropriate update ID to use for a record for each write.

Data Interfaces

Sirocco provides the following types of operations:

- **Write:** Given a data buffer, container ID, object ID, fork ID, start record, number of records, record length, and update ID, the range will be overwritten on the server transactionally (i.e., all data is written to the record range, or none of it is). Optionally, one can provide an update

condition (see Section) and an update ID to use with it. A user can omit the update ID, and instead use an automatically incremented update ID on that store.

- **Read:** Sirocco supports sparse data within forks, creating a need for obtaining a map that describes the data present. The read operation allows for the user to obtain the map, the data within the requested range, or both as an atomic operation. The map will, for all records present, indicate the size and update ID of each record.

  Any read operation requires an container ID, object ID, fork ID, start record, and number of records. A buffer is required for the map and/or the data to be received. Optionally, one can provide an update condition and an update ID to use with it. One can provide invalid IDs for the fork, object, and/or container ID (called INVALID_ID in the API) to obtain a map of an object, a container, or containers, respectively, within the store.

An interesting feature of this API is that there are no "create" or "delete" operations. Creation is implicit: Writing a range of records in an object is enough. One way to think about this is to reason that all objects exist at all times, they just do not have any data within them. Likewise, deletion is resetting the records within the object to their default state. This is accomplished with a reset call, which is not yet developed enough to discuss in this document. A command similar to punch can be implemented with a write of zero-length records over a range. Note that this does not have a distributed effect, so any copies of the data residing on other servers will continue to persist.

Data Durability

One side effect of the free placement enabled by the Sirocco model is that data can be held temporarily in a location that is not ultimately considered safe enough to hold the data. This is a typical behavior in a write-back caching model. For instance, In a local POSIX file system, bulk I/O contents can be temporarily held in memory. The user is only guaranteed that I/O requests will reach full durability if she calls "sync." Sirocco employs a similar technique.

Each server has a durability attribute that it holds for a fork. This durability attribute is an abstract value that acts as a measure of final durability the contents of that fork should have. These values are satisfiable by storing data on sufficiently durable stores, e.g. a set of disks in a RAID, or by replicating across less safe stores. The server accepting the write is obligated to eventually ensure that the data is protected to the specified level of durability at some point in the future. The client ensures that this has taken place by issuing a sync command over a range of records.

Batched Updates

To increase network efficiency, it is possible to send several commands to a server at once. This mechanism can be used to implement list I/O [2] or other non-contiguous I/O operations, even across different objects. Sirocco also allows batches to be specified as transactional, enabling fully

ACID updates to records. Paired with concurrency control (see Section), transactional batches can be an extremely powerful construct.

Concurrency Control

Concurrency control is the ability to ensure a correct outcome to concurrent updates to shared data by multiple clients. There are two methods for accomplishing concurrency control. The most common used in file systems is pessimistic concurrency control, where explicit locks are used to protect data. Other systems, including database management systems, employ optimistic concurrency control, where operations can be attempted, then rolled back in the event of an invalid concurrent modification [5]. Because there are benefits to each approach in different situations, Sirocco has facilities for each type.

Optimistic Concurrency Control
Optimistic concurrency control is beneficial in cases where the likelihood of conflicting operations is low, and locking would incur a significant overhead on the operation. An example is a read-modify-write operation on a remote store. Taking and releasing a lock would double the number of network round trips to complete the operation.

Sirocco implements optimistic concurrency control for a more general workload. Instead of inspecting data, the server executing the operations compare the incoming update IDs with those already present. If the transaction contains a *conditional write*, the server executing the write will ensure that the incoming update ID is greater than any currently present.

Conditional writes work natively within transactional batches, extending their applicability to more complex workloads. If a conditional write fails, the enclosing transaction is also failed, rolling back any changes previously incurred. More information on uses and performance of conditional updates can be found in previously published work [1]

Pessimistic Concurrency Control
Sirocco also provides storage semantics that a client library can leverage to implement traditional leased locks. The basic mechanism is the triggered batch, an operation that is similar to a conditional update, but with three important differences. First, a triggered batch does not fail when a condition is not met, but instead is queued and executed in order once the condition becomes true (probably due to a non-triggered write from another server). Second, a triggered batch is simply a condition attached to a batch that will execute in full once the condition is satisfied; it does not modify data on its own. Third, a triggered batch can only be used on a single record.

A few additional considerations are allowed for triggers to enable failure recovery if a client fails to release a lock. During a triggered operation, the client is notified of progress: First when the operation is deferred for later execution, and then when the queued operation is next in line to be executed (along with a server time stamp). This allows a waiting client to detect that the current holder of the lock is not releasing it, thus allowing the client to invoke a recovery protocol.

We have sketched a simple leased locking protocol using triggered batches.

1. A lock requester will submit a triggered batch on the update ID of a record on the lock server, allowing the operation to proceed when the update ID reaches a value indicating an unused state. The batch will contain a write operation in addition to the trigger, setting the lock record's update ID to a value indicating the lock is in use (say, two), its value to the relative duration of the lock.

2. When the requester is at the head of the wait queue on the record, the server will notify the locker of its status, along with the time the last conditional batch executed, i.e., the start time of the last lock.

3. The requester can then issue a read to get the duration of the lock from the record and reconcile that with the time received in the head of queue notification to calculate the duration of wait.

4. The client currently holding the lock, as part of the release, will write the record such that the update ID is one, allowing the requester's trigger to be satisfied. The requester's triggered batch will execute, modifying the record to reflect the requester obtaining the lock.

5. Once the requester finishes using the lock, the requester will release the lock by issuing a conditional write to the record resetting the update ID to one, allowing the next (if any) triggered batch to execute.

There are two failure cases to consider in this protocol: the non-responsive locker, and the non-responsive requester at the head of the queue. In the case of the non-responsive locker, the requester at the head of the queue is able to force the non-responsive locker out after reading the record and its metadata by issuing a batch conditional on the previous locker's version. If the non-responsive locker is simply slow to respond, when it attempts to unlock the record, it will receive a deferral in the execution of its lock release, allowing it to cancel gracefully. The non-responsive requester can be removed from the head of the queue without issue, as it never receives a head-of-queue message.

Read/Location Protocols

While free data placement helps some workloads, reading requires extra work. While we do not consider read-heavy workloads a solved problem, Sirocco provides two facilities to make reads possible.

The first facility enables reading data that the client did not predict it would need, i.e. a random, one-off read. Such a read requires an extensive search over the population of storage servers, which Sirocco will perform on behalf of the user. We are conducting research on how best to reduce the expense of such searches [9], improving the efficiency over exhaustive broadcast searches.

The second facility, proxying, enables efficient reading in the case of data that we can predict will be read, especially by multiple clients. Data like file system structures and metadata can be quite efficiently supported through this mechanism. It is noteworthy that this mechanism relies on cooperation of clients that would access the proxied range of records.

Each server, when initially started, considers itself non-authoritative for all records. This quality simply means that any read requests that are issued to that server may be anywhere in the system, or may have been recently written to a new server, so no optimizations to the location process are possible. However, by convention with other clients, a server may be deemed authoritative for particular read requests by setting a flag. Once the server is authoritative for that range, clients must direct all write requests for that range to that server. This enables caching of data and/or previously discovered locations, enabling correct proxied access to data so long as the server remains authoritative. Although cached locations of data may change due to migration, this is easily detected by the authoritative server by inspecting update IDs of read data.

Conclusion

Sirocco is a paradigm shift in parallel storage system design. While there are many opportunities for improved performance, Sirocco remains a work in progress. As we develop and use Sirocco, we will better understand the requirements for efficient client operations. We have already identified several facilities to enable efficient file system client implementations.

As Sirocco is developed and refined, we will be making updates and corrections to this document. Future iterations will be available at `http://www.cs.sandia.gov/Scalable_IO/sirocco`.

References

[1] P. Carns, K. Harms, D. Kimpe, J.M. Wozniak, R. Ross, L. Ward, M. Curry, R. Klundt, G. Danielson, C. Karakoyunlu, J. Chandy, B. Settlemyer, and W. Gropp. A case for optimistic coordination in HPC storage systems. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 48–53, Nov 2012.

[2] Avery Ching, A. Choudhary, Wei-Keng Liao, R. Ross, and W. Gropp. Noncontiguous I/O through PVFS. In *Cluster Computing, 2002. Proceedings. 2002 IEEE International Conference on*, pages 405–414, 2002.

[3] Matthew L. Curry, Ruth Klundt, and H. Lee Ward. Using the Sirocco file system for high-bandwidth checkpoints. Technical Report SAND2012-1087, Sandia National Laboratories, Feb 2012.

[4] C. Karakoyunlu, D. Kimpe, P. Carns, K. Harms, R. Ross, and L. Ward. Toward a unified object storage foundation for scalable storage systems. In *Cluster Computing (CLUSTER), 2013 IEEE International Conference on*, pages 1–8, Sept 2013.

[5] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, June 1981.

[6] Zhenyu Li, Zengyang Zhu, Zhongcheng Li, and Gaogang Xie. SAP2P: Self-adaptive and locality-aware P2P membership protocol for heterogeneous systems. In *Parallel, Distributed and Network-Based Processing, 2008. PDP 2008. 16th Euromicro Conference on*, pages 229–236, Feb 2008.

[7] R.A. Oldfield, L. Ward, R. Riesen, A.B. Maccabe, P. Widener, and T. Kordenbrock. Lightweight I/O for scientific applications. In *Cluster Computing, 2006 IEEE International Conference on*, pages 1–11, Sept 2006.

[8] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '01, pages 149–160, New York, NY, USA, 2001. ACM.

[9] Zhiwei Sun, Anthony Skjellum, Lee Ward, and Matthew L. Curry. A lightweight data location service for nondeterministic exascale storage systems. *Trans. Storage*, 10(3):12:1–12:22, August 2014.

DISTRIBUTION:

1	MS 1319	Matthew L. Curry, 1423	
1	MS 0899	Technical Library, 9536	(electronic copy)

This page intentionally left blank.


Sandia National Laboratories